PCT

WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 6 : **H04Q 11/00**	**A2**	**(11) International Publication Number:** **WO 98/59519** **(43) International Publication Date:** 30 December 1998 (30.12.98)

(21) International Application Number: PCT/CA98/00537

(22) International Filing Date: 2 June 1998 (02.06.98)

(30) Priority Data:
08/880,459 24 June 1997 (24.06.97) US

(71) Applicant: NORTHERN TELECOM LIMITED [CA/CA]; P.O. Box 6123, Station A, Montreal, Quebec H3C 3J5 (CA).

(72) Inventors: OLDER, William, Joseph; R.R. # 3, Merrickville, Ontario K0G 1N0 (CA). BIERMAN, Eric; 1701–71 Somerset Street W., Ottawa, Ontario K2P 2G2 (CA).

(74) Agents: BRETT, R., Allan et al.; Smart & Biggar, 900–55 Metcalfe Street, P.O. Box 2999, Station D, Ottawa, Ontario K1P 5Y6 (CA).

(81) Designated States: CA, JP, European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE).

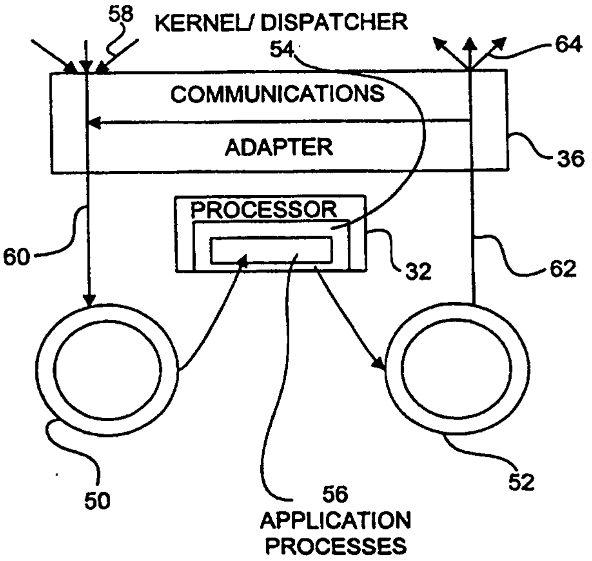
Published
Without international search report and to be republished upon receipt of that report.

(54) Title: ASYNCHRONOUS MESSAGE PROCESSING SYSTEM AND METHOD

(57) Abstract

A new asynchronous message processing system and method are provided in which a process is completely defined by a state and a type. Incoming messages received by a network node are stored in an input message queue and are handled in sequence in the order they arrive, and outgoing messages are stored in an output message queue and are transmitted in sequence in the order they are generated to a destination network node. Incoming messages addressed to a process are handled by invoking a particular message handling application code which runs to completion, and which may modify the process state. Process state modifications are atomic in the sense that they either completely happen or do not happen at all; partially implemented process state modifications are undone.

OVERALL MESSAGE FLOW



FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AL	Albania
AM	Armenia
AT	Austria
AU	Australia
AZ	Azerbaijan
BA	Bosnia and Herzegovina
BB	Barbados
BE	Belgium
BF	Burkina Faso
BG	Bulgaria
BJ	Benin
BR	Brazil
BY	Belarus
CA	Canada
CF	Central African Republic
CG	Congo
CH	Switzerland
CI	Côte d'Ivoire
CM	Cameroon
CN	China
CU	Cuba
CZ	Czech Republic
DE	Germany
DK	Denmark
EE	Estonia
ES	Spain
FI	Finland
FR	France
GA	Gabon
GB	United Kingdom
GE	Georgia
GH	Ghana
GN	Guinea
GR	Greece
HU	Hungary
IE	Ireland
IL	Israel
IS	Iceland
IT	Italy
JP	Japan
KE	Kenya
KG	Kyrgyzstan
KP	Democratic People's Republic of Korea
KR	Republic of Korea
KZ	Kazakstan
LC	Saint Lucia
LI	Liechtenstein
LK	Sri Lanka
LR	Liberia
LS	Lesotho
LT	Lithuania
LU	Luxembourg
LV	Latvia
MC	Monaco
MD	Republic of Moldova
MG	Madagascar
MK	The former Yugoslav Republic of Macedonia
ML	Mali
MN	Mongolia
MR	Mauritania
MW	Malawi
MX	Mexico
NE	Niger
NL	Netherlands
NO	Norway
NZ	New Zealand
PL	Poland
PT	Portugal
RO	Romania
RU	Russian Federation
SD	Sudan
SE	Sweden
SG	Singapore
SI	Slovenia
SK	Slovakia
SN	Senegal
SZ	Swaziland
TD	Chad
TG	Togo
TJ	Tajikistan
TM	Turkmenistan
TR	Turkey
TT	Trinidad and Tobago
UA	Ukraine
UG	Uganda
US	United States of America
UZ	Uzbekistan
VN	Viet Nam
YU	Yugoslavia
ZW	Zimbabwe

ASYNCHRONOUS MESSAGE PROCESSING SYSTEM AND METHOD

Field of the Invention

The invention relates to a distributed asynchronous message processing system and method.

Background of the Invention

Various models of communicating processes have become popular for the specification and implementation of distributed systems. Most of the well known systems, from ADA's Rendezvous to the Remote Procedure Call, as well as most of the formal calculi are based on a synchronous model of communication in which a sender suspends until a reply is received by a suspended receiver. Such systems have the great practical advantage that since remote communications can be hidden behind a procedure call, an existing local application can be made over into a distributed one with minimal impact.

However, the synchronous model has several problems in essentially distributed reactive systems or command-and-control systems. Any system in which processes block, waiting for specific events, is subject to communication deadlocks. In general, such deadlocks can be analysed for only the simplest of systems (a fixed number of finite state machines with very few states and rigid communication patterns) since it depends on the global state of the system. Stringent organization of the communication as a client/server hierarchy may prevent deadlocks but forces an often clumsy organization in which active objects (spontaneous message sources) can never communicate directly with each other. In particular, the handling of peer protocols and the handling of exception and failure conditions run into problems, as they involve message flow in the opposite direction to the normal pattern.

Finally, with high speed RISC (reduced instruction set computer) processors, the simple state transitions caused by arriving messages and the transmission times for short messages which make up the bulk of a command-and-control system require on the order of microseconds of CPU time, while the delays between distant nodes of a network are measured in milliseconds and round-trip times often in tens of milliseconds. The ratio of the message transmission time between distant nodes to the local message processing time is referred to as a "time constant ratio". The time constant ratio for the time periods outlined above is roughly 10^3 - 10^4, and is already quite high. Since the round trip delay is dominated by propagation delays, depending essentially on the speed of light, it will not be decreasing, and will likely generally increase by up to another factor of 10 with the growth of more global networks. The processing times and transmission time, however, can be expected to go down by at least a factor of 10 in the next few years, resulting in the potential for time constant ratios as high as 10^6.

The time constant ratio is a critical parameter characterizing distributed systems. We know from personal daily experience that we are usually happy to spend 5 seconds making a telephone call for a 5 minute conversation (r=1/60), but we would be put out if making the call took 5 minutes (r=1), and justifiably angry to spend 5 minutes waiting for a 5 second conversation (r=60). For r>1, it makes sense to suspend the waiting process, and do something else (if context switch costs are not too high), and then r provides an idea of how many suspendible "threads" are needed to keep the processor busy. In the current world of LANs (local area networks) and PCS (personal computers) running slightly distributed applications (with r typically 10 - 100), the synchronous communication model is viable. However, with r=1000, let alone $r=10^6$, it is no longer appropriate.

One problem is that since suspension must save the execution stack, each suspendible process or thread must be allocated stack space sufficient for its worst case, with values typically being in the range of 10 Kbytes to 1 Mbytes. For systems in which the natural concurrent "objects" are relatively numerous, on the order of r, this results in excessive or infeasible memory demands. A second problem is that the suspension overheads are themselves longer than typical transition times in the current application.

Summary of the Invention

It is an object of the invention to obviate or mitigate one or more of the above identified disadvantages.

According to a first broad aspect, the invention provides in a network node having a processor, and a memory, an asynchronous message processing method comprising the steps of: maintaining a plurality of processes in the memory, each process consisting of a

process state and associated application code, each process being identifiable by a pid (process identifier); the node receiving incoming messages;
for each incoming message received, the processor determining from a destination pid forming part of the message to which of said plurality of processes the message is addressed, and running the application code associated with the destination pid to handle the message as a function of the destination process's process state and making changes to the destination process's process state if necessary; the application task, while running, generating outgoing messages if necessary;
the node transmitting any outgoing messages.

According to a second broad aspect, the invention provides a network node for connection to a digital network comprising: an interface to the digital network for receiving incoming messages and storing them in an input message queue, and for reading outgoing messages from an output message queue and transmitting them; a processor for maintaining a plurality of processes in a memory, each process consisting of a process state and associated application code, each process being identifiable by a pid (process identifier), and for reading messages one at a time from the input message queue in the order they were received, determining from a destination pid forming part of the message to which of said plurality of processes the message is addressed, and running the application code associated with the destination pid to handle the message as a function of the destination process's process state and making changes to the destination process's process state if necessary and writing an outgoing message to a next available buffer location in the output message queue if necessary.

According to a third broad aspect, the invention provides a digital network comprising a plurality of network nodes, each network node comprising: an interface to the digital network for receiving incoming messages and storing them in an input message queue, and for reading outgoing messages from an output message queue and transmitting them;

a processor for maintaining a plurality of processes in a memory, each process consisting of a process state and associated application code, each process being identifiable by a pid (process identifier), and for reading messages one at a time from the input message queue in the order they were received, determining from a destination pid forming part of the message to which of said plurality of processes the message is addressed, and running the application code associated with the destination pid to handle the message as a function of the destination process's process state and making changes to the destination process's process state if necessary and writing an outgoing message to a next available buffer location in the output message queue if necessary.

Brief Description of the Drawings

Preferred embodiments of the invention will now be described with reference to the attached drawings in which:

Figure 1 is a block diagram of a simple

network;

Figure 2 is a block diagram of one of the nodes in the network of Figure 1;

Figure 3 is an illustration of the overall flow of messages within a node, according to an embodiment of the invention;

Figure 4 is an illustration of a shared input buffer ring;

Figure 5 is a chart of a representative message syntax;

Figure 6 is a summary of kernel data structures and environment registers;

Figure 7 depicts an undo log (trail) mechanism; and

Figure 8 is a flow chart of the kernel loop.

Detailed Description of the Preferred Embodiments

Network Context

Referring firstly to Figure 1, a network context for implementing the system and methods according to an embodiment of the invention is comprised of a number of processing nodes 10 interconnected by a digital communications network 20. The communications network 20 may represent a local area network, a wide area network, even a global network, for example. The network 20 must be capable of sending digital messages, up to some modest maximum size, from any processing node 10 to any other processing node. The network 20 may be either a connection oriented or connectionless medium, and should provide an acceptably low rate of message loss, corruption or out of order deliveries. Many communication technologies could be used or adapted for use by means of appropriate lower level protocols, but as a representative concrete example we will assume the use of single cell ATM (asynchronous transfer mode). In this case we will assume that a single virtual channel exists between each pair of nodes 10 (in each direction), and that there is in each node a table which maps node identifiers into local virtual channel identifiers. It is also assumed that the quality of service provided has a low enough rate of lost or corrupted messages that no higher level protocol is necessary.

Node Structure

Referring now to Figure 2, the individual nodes 10 minimally consist of a high speed local bus 30, a processor or cpu 32, some random access shared memory 34, and an intelligent communication adaptor 36 which provides access to the communications network 20 (see Figure 1). The communication adaptor 36 and bus 30 are such as to allow direct memory access (DMA) of data between the communication adaptor 36 and message buffers in the shared memory 34, and such memory buffers are visible to the processor 32. In the preferred implementation, normal communications between the processor 32 and the communication adaptor 36 are handled through the shared memory 34 only and no interrupts are generated by the communication adaptor 36 except possibly under exception conditions.

In addition to a virtual channel between any pair of nodes, there is a "looped back" channel from each node back to itself, allowing a node to send itself messages. This may be done in the communication adaptor

36, emulated in the processor 32, or done via the communications_network, for example.

Each processor is executing a run-time environment according to an embodiment of the invention and some application code directly, without any standard OS (operating system) intervening. It is noted that a run-time environment may run in a conventional OS context, but this complicates the description considerably.

Overview of Run-time Environment

This example implementation of the invention will be further described by considering the operation of a single node 10. The run-time environment in the processor 32 is controlled by a kernel which runs on the processor 32 periodically or continuously. It is noted that the "periodic" case is one method of operation which may be used in an OS context, but that continuous execution is preferable and this will be assumed in the description which follows.

Each processor 32 also runs application "processes" which are invoked by the kernel. Each process has a lifetime from the time it is created by another process until it voluntarily terminates. When we use the term "process" it is not in the conventional sense of a continuous stream of application code which is run concurrently with other processes by time slicing the processor time and saving a stack between time slices allocated to the particular process. Instead, a "process" is defined by 1) a process state consisting of a set of state variables stored in shared memory 34 owned by the process which exist from when the process is created until it terminates and 2) a process type defined by the code which defines the behaviour of the process and which operates on the state variables to cause state transitions. To reiterate, the only thing which exists continuously during the life of a process is the process state and the process type definition.

The kernel is continuous in the sense that it is started and never terminated. The kernel controls when application processes are run. The kernel calls the application processes directly and thus in this sense is always running.

Broadly speaking, (by way of overview), the kernel takes messages received by the particular node one at a time, determines for which destination process the message is intended, runs to completion the code identified by the process type of the destination process, possibly resulting in one or more changes to state variables belonging to the process state, and possibly outputting one or more messages to be sent to other processes, and finally returns to the kernel. This sequence of steps taken to handle each message is considered to constitute one "state transition", or simply one "transition".

The kernel does not process the next message received by the particular node until the state transition associated with the previous message is complete. In fact, the kernel does nothing until a return from the application code run for that transition occurs. While the application code invoked in association with a particular process is running, the process is in an "Active" mode. When the application task is complete, the process is in a "Inactive" mode. While Inactive, the process state and type continue to exist, but the processor is not running any application code associated with the process and no changes to any state variables belonging to that process occur. Because transition application code runs to completion with no time slicing or intervention by other processes, there is no need to ever perform context switches between processes.

It is a preferred feature of the invention that state transitions be atomic. By "atomic", it is meant that either all of the operations associated with a state transition occur, or none of them occur. Thus if for some reason, not all of the operations associated with a state transition can be completed, then the effects upon the state of any operations which have been completed need to be rolled back.

The behaviour of each process is uniquely determined by its process state and process type. The states of different processes are non-overlapping. The process state of a process can only be changed by its receiving and processing a message. The state transitions of a process are serialized and form a strict sequence. The process exclusively "owns" (and is responsible for) its state data and storage. The application code associated with the process type specifies completely the state changes and messages sent for each sort of message received. Returning successfully from this code completes the state transition. Between transitions each process is in Inactive mode waiting for its next message.

When the application code for each transition completes, it returns a return code to the kernel. The return code is either "commit" indicating that the code was successfully executed, or "abort" indicating that the code was not successfully executed. In the case of a commit return code, a successful state transition has occurred. The kernel will perform some "post-transition" processing before moving on to the next incoming message, as discussed in further detail below.

Process Creation, Termination and Identification

Any given process "P" is created as a byproduct of a state transition of some preexisting process "A" on the same processor. P continues to exist as long as it chooses to, independently of A. Any given process terminates by modifying its commit return code on receiving its "last" message. This modification is detected by the kernel and processed during its post-transition processing. At this time, the state storage allocated to the terminating process is returned to the appropriate memory pool, the process type is returned to a default type which causes all messages to be discarded, and the incarnation number is incremented so that any outstanding messages to the terminated process are discarded. Both process creation and termination are part of the atomic transitions; in particular, creation of P is part of some atomic transition of its creator A. Direct creation can only be done locally (in the same shared memory) so that to get a remote process created, a process must work indirectly through a preexisting agent which may impose whatever security restrictions may be appropriate.

Each process once created has a destination pid (process identifier) which is used by other processes to address messages to the process. Destination pids are discussed in detail below. An initial connection protocol is needed to make destination pids globally available. Initially the only processes that know the pid of a process P are P itself and its local creator. The normal way to get a pid of a remote process is to send an initial request to some well-known "name server" process, the reply to which will contain the desired pid. This could be for either a pre-existing or a newly created process.

The process names of well-known name server processes (at least one on each node) can be handled in two ways which are preferably used in combination: The first is to give distinguished ubiquitous servers each a fixed index (a fixed or no incarnation number) on all nodes. The second is by a generic distributed name service based on having the well-known processes register themselves (the name server itself using the first strategy).

Overall Message Flow

Messages are the sole means of communication from process to process, regardless of whether the processes are on the same node or not. The overall flow of messages through a single node is shown in Figure 3. Each node has a rotating input buffer ring 50 and a rotating output buffer ring 52, both forming part of shared memory 34 (see Figure 2). The processor 32 runs the kernel 54 which performs a dispatcher function among other functions, and which reads from the input buffer 50, and controls the execution of application processes

56. The application processes 56 write to the output buffer 52.

Ordered input streams 58 from all other nodes are merged into a single serial input stream 60; in the case of ATM this merging occurs somewhere in the ATM network itself, but could in other circumstances occur in the adaptor 36 as illustrated. Incoming messages are inserted into the next free slot of the rotating input buffer ring 50. The kernel 54 asynchronously pulls messages from the input buffer ring 50 and handles them in order of arrival by activating the appropriate application process 56 without the requirement for any scheduling of processes. The activation of the appropriate application process 56 is described in further detail below. Outgoing messages go into consecutive slots in the rotating output buffer ring 52, from where they are pulled in order by the adaptor 36 and placed in a single serial output stream 62 which will contain ordered streams 64 to all other nodes. It is possible that the adaptor may reorder messages between various streams, but not within a single stream.

The rotating input buffer ring 50 and the rotating output buffer ring 52 use a rotating ring mechanism to provide automatic recycling of free buffers without requiring any explicit intervention by the kernel 54. The size of the rings allows for asynchronism between input process, code execution, and output process, allowing for fluctuations in transition execution rate. The load on the system can be simply characterized by the occupancy of the buffer rings 50,52. The sizes of the buffer rings 50,52 must be engineered large enough that loss rates due to buffer overruns are acceptably low (i.e. on the same order as other loss mechanisms.)

Input Buffer Ring

The details of records or individual buffers in the rotating input buffer ring 50 will be described with reference to Figure 4. The rotating input buffer ring 50 is a data structure in shared memory 34 for handling incoming messages from the communications adaptor 36. The rotating input buffer ring 50 is preferably a statically configured (e.g. at node initialization time) ring (i.e., a circular chained list) of fixed (maximum) size message buffers. The use of a static ring structure avoids the need for dynamic list manipulations during processing, as well as mutual exclusion semaphores, and explicit buffer management recycling. Alternatively, buffers may be added to or deleted from the rings dynamically by the processor to adjust for slow changes in occupancy statistics.

Each individual message buffer contains the address of the next message buffer in the chain, these addresses collectively defining where the ring is stored in memory. In a statically configured ring, these addresses are constant. The fields in each message buffer (excluding the address of the next buffer) are shown in Figure 4. Each message buffer has a status word 70 used for synchronization as described below, possibly some additional control information (such as virtual channel identifiers in the case of ATM), and each message buffer has space to store the body 72 (or content) of a message. The specific order and content of the fields in each buffer may be set up to suit any particular messaging protocol/buffering scheme.

In a variant implementation (not shown), the message bodies and some or all of the dynamic control/status fields may be in a different portion of memory and a pointer to the body only is in the buffer ring. This variant is useful in systems in which virtual memory addresses used by the processor are different from the physical memory addresses used by the adaptor, as it permits each to have its own static ring structure pointing (in different memory address spaces) at the same (unlinked) message buffers.

As shown in Figure 4, the body 72 of a message includes a destination pid field 74, a source pid field 76, and a data field 78. Messages are addressed to individual processes by having the corresponding pid as their destination pid field 74. The specific size and structure of the destination and source pid fields 74,76 is determined by the addressing needs of each specific instance of the network. The general structure of the pid field includes a node identifier 80 to be used to identify the node with which the process is associated, a local pid 82 to identify the process on each node, and an incarnation index 84 to be used as an incrementing counter which is incremented each time a process having a particular local pid 82 is created. The incarnation index 84 is provided to limit the rate of pid reuse and to provide sparseness. A particular incarnation of a process type is a process instance.

A process instance table holds a mapping from the local pid 82 of the destination pid to the information needed to execute application code associated with the indicated process. For each local pid 82, there is a record 83 in the process instance table. The record consists mainly of a code pointer 86 to the location in shared memory storing the particular application code which is to handle a message for the process instance, and a state pointer 88 to the state information of the process instance. The incarnation index of the pid is also kept in a field 90 of this record. There may also be a pointer to a structure containing generic attributes of the process type. In addition there may be other (per instance) attributes associated with debugging control or performance measurements. These may differ between various implementations and modes of execution (e.g. simulation versus development versus field contexts).

Output Buffer Ring

The output buffer ring is analogous to the input buffer ring.

Input Buffer Operation by Adaptor

A function of the communications adaptor 36 is to transfer (via DMA) each valid (i.e. not "empty" cell) received message into the buffer in the input buffer ring 50 pointed to by a state variable input_write_head, for example. The state variable input_write_head, is used to point to the next empty buffer in the input ring buffer 50. This state variable may be private to the adaptor 36 or alternatively may be kept in shared memory 34, but is (apart from initialization) only read and updated by the adaptor 36. After each successful message transfer the adaptor 36 sets a flag in the status field in the current buffer to indicate that the buffer is occupied with a valid message, and advances input_write_head to the next buffer in the chain. Another state variable input_read_head, for example, is maintained by the kernel 54 to control reading messages from the input buffer ring, as discussed in further detail below.

Output Buffer Operation by Adaptor

Another function of the communications adaptor 36 is to transfer (via DMA) each valid (i.e. not "empty" cell) message written into the output buffer ring 52 onto the outgoing communications stream 62. A state variable, for example output_read_head, is used to point to the message to be read from the output buffer ring 52. This state variable may be private to the adaptor 36 or alternatively may be kept in shared memory 34, but is (apart from initialization) only read and updated by the adaptor 36. After each successful message transfer the adaptor 36 sets a flag in the status field of the current buffer to indicate that the buffer is available again, and advances output_read_head to the next buffer in the chain. Another state variable output_write_head, for example, is maintained by the kernel 54 to control writing messages to the output buffer ring 52, as discussed in further detail below.

Message Syntax

A representative message syntax is shown in Figure 5. As indicated above, it includes the destination 74 and source pids 76 at the front of each message. The source pid 76 is stamped automatically on each message by the kernel 54 of the run-time system, and the destination pid 74 is inserted, also by the kernel, as part of the message sending primitive operation. The application code 56 does not have direct access to either of these fields of a message, but can ask for the message source pid as an explicit call.

The rest of the message contents (the data field 78) is the concern of a message marshalling/demarshalling apparatus, which ideally may be integrated with the application programming language. The details of this are not covered here. Typically it encodes methods 92, format indications 94 (such as the number of arguments 96 and a compressed type signature), a standard machine-independent representation of basic types (including pids) and some sort of checksum 98. This format may optionally be encrypted.

Kernel Data Structures and Environment Registers

Figure 6 illustrates how the run-time system might be structured in a particular implementation. With this implementation, the kernel maintains a system stack 100 and a trail stack 102 (discussed below) and a series of environment registers as follows:

mi - "message in" = input_read_head : provides access to message source and destination pids for messages in the input buffer ring 50; **mi**.status is true when a message is present, and **mi**.dest pid is the destination pid contained in a message;

in - input byte stream : used by demarshalling code to extract message fields from messages in the input buffer ring 50;

no - "next out" = output_write_head : next available output buffer used to set destination, source pids in messages in the output buffer ring 52;

out - output byte stream : used by marshalling code to format outgoing messages in the output buffer ring 52;

pe - process entry : provides access to all process instance and type attributes through the process instance table 85;

sp - stack pointer : top of system stack 100;

te - trail stack end : end of undo log trail stack 102; and

pc - program counter : points to next executable instruction in the executing code.

The variables associated with the input and output buffer rings have been discussed above. The main point of interest about the system stack and trail stack is that there needs to be only one in each processor in the overall system.

Also shown are a memory pool 104 which is an area of memory 34(see Figure 2) reserved for process state storage, a code memory 106 which is an area of shared memory 34 for storing application code. An active timer list 112 (managed by the kernel) is also located in the shared memory 34.

The code pointer 86 is shown pointing from the record in the process instance table 85 pointed to by **pe** to the location in the code memory 106 where the application code is stored. The state pointer 88 is shown pointing from the same record in the process instance table 85 to the location in the state memory 104 of the process state for that process instance. The process state may include one or more timer pointers 114 to timers in the active timer list 112.

The Undo Log Mechanism

There are several ways of implementing atomicity of state transitions. Under the assumptions that most messages update only a very small amount of state, and that the actual need to restore an old state is very rare, then an efficient software implementation can be based on logging the old values of any changed state variables, as illustrated in Figure 7.

For this purpose a single system resource called the trail stack 102, consisting of an array (or stack) of (address, value) pairs, of sufficiently large size is needed. Initially, and just before every application task execution, the environment register **te** (trail end) is set to point to the start 103 of the trail stack 102. Every write to a word of state memory during application task execution must first push the address (addr) and current value (oldval) of that location onto the trail stack 102. (The efficient implementation of this requires support from the language compilers and code generators.) When a transition is committed (i.e. the application task successfully completes and the process returns to a suspended mode) **te** is reset to the top of the trail stack 102, ready for the next message.

When a transition needs to be aborted, the trail is processed in reverse order (from **te** backwards), with the old values being restored to the locations indicated in the process state memory 104, for example with code along the lines of:

```
while (te>trailstart) {
 val -trail(te); te--;
p  ←trail(te); te--;
         (p)*←val; }
```

The new values (newval) written during the unsuccessful transition are discarded.

Trailing applies to all updates of state variables: these consist of those in the memory segment pointed to by the state pointer of a process and everything reachable from there. It also includes certain changes in the timer area (associated with creation, resetting, and deletion of timers being used by the process; timers are discussed in more detail below), the process instance table (associated with creation of processes), and memory pools (associated with dynamic memory allocation). In this fashion, all timer operations, process creations, and memory allocations will automatically be included in the atomic action.

Trailing is *not* needed for updates to the stack or output message buffers, or other volatile variables. Also, as an optimization, it may not be necessary for the initialization (constructor) code for state objects.

It is of course to be understood that other mechanisms for achieving atomicity may be used without departing from the scope of the invention.

The Kernel Dispatcher Process

The processor 32 runs the kernel between running application code. The kernel has a main loop which is run once per incoming message. The kernel loop is shown as a flow chart in Figure 8. The kernel loop will be described in the context of the previously introduced kernel data structures and environment registers which were described with reference to Figure

6. As mentioned previously, the kernel has an internal state variable input_read_head (or **mi**) which points into the next available message in the input buffer 50. It runs an idle task as long as the input buffer is empty as indicated by the status register **mi**.status being equal to zero (block 200, yes path, and block 202). The function of the kernel is to process incoming messages as fast as possible in their order of arrival. When an incoming message arrives (block 200, no path) and the status flag field of the message buffer pointed to by input_read_head indicates a valid message, the kernel extracts the message's destination pid, **mi**.dest, extracts the local pid from the destination pid, verifies that the local pid is in the correct range, and uses it to access a record in the process instance table 85. A field of the process instance record contains the incarnation index: this must match the destination pid from the incoming message exactly. If the extracted incarnation index is out of range, the incoming message is deemed to be invalid (no path, block 204), an error counter is incremented (block 206), and the message is discarded by simply clearing its status field and advancing the input_read_head pointer (block 222).

If the message pid is valid (yes path, block 204), the kernel then sets up the environment variables for process execution and atomicity support and invokes the application code pointed to by the code pointer in the process instance table. One such (notional) environmental variable may provide the message source pid, in case the application code needs to access it.

(However, as applications rarely need to ask for the source of a message, it is sufficient and more efficient to merely copy a pointer to the message source field to a hidden environment variable (possibly a register), and to use an indirect memory reference to extract the source field if needed by the application.) Similarly, a pointer to the process instance table record is left in an environment variable (possibly a register), in case the application code requests information available in that place (e.g. the pid, or process type attributes.)

To invoke the application code, the state pointer 88 and the code pointer 86 are read from the process instance table 85 and fetched into registers and/or pushed onto the stack, depending on the call conventions. The method or command field of the message must be extracted, and the pointer to the next field of the message after the command, namely the input byte stream register **in,** is also fetched into a register and/or pushed onto the stack. For atomicity support on outgoing messages, a copy (local to the kernel) of the output_write_head is made (block 210). The application code then runs (block 212) by invoking the application code pointed to by the code pointer, passing the input byte stream **in** and the state pointer as parameters, and expecting a return code or result “res”.

When the application code returns to the kernel, its return code res (either in a register or at the top of the stack by convention) indicates either "commit" or "abort". If the code indicates that the decision is to abort (no path, block 214), the kernel resets output_write_head from its local saved value, thus cancelling any outgoing messages. Then it needs to restore the initial state of the process by processing its undo trail as described previously (block 216). Then the input message is discarded by clearing its status field and advancing the input_read_head to the next message (block 222).

If the return code indicates "commit" (yes path, block 214), any outgoing messages which were generated by the application code, starting with the saved value of output_write_head to the current value of output_write_head, must have their status fields set, so they will be processed by the communication adaptor. The trail can be reset simply by assigning trail_end := trail_start (block 218). In the special case that the process is to terminate, the pid and process state associated with the process are deallocated (block 220). Finally, the input message is discarded by clearing its status field and advancing the input_read_head to the next input message (block 222).

If any software exception (arrow 223) occurs during execution of application code, appropriate diagnostic information can be captured (block 224), the system stack is reset to its normal point (ie. where it would be just before the call to the application code), and control transferred to the abort processing (block 216) described above. Note that relevant diagnostic information, roughly in order of increasing size and decreasing utility, consists of: (0) the error type and program counter, (1) the current input message, (2) the (complete) current process state, and (3) the complete input buffer ring (which includes both recently processed messages and the near term future messages).

As an example of a software exception, for each type of process there may be a maximum allowed time for transitions of processes of that type. If a transition exceeds this limit, an exception will be generated and handled as described above.

Note that in the implementation described here there is no difference at all in processor overhead which depends on whether a message is external or purely internal. If internal messages are optimized to eliminate the implicit copying from output buffer to input buffer, this difference becomes (at most) the DMA cost of the copies. An optimized (assembler language) implementation of the kernel loop as described above requires only about 50 machine instructions, which makes the kernel overhead a fraction of a microsecond in typical workstations.

A process transition may generate an outgoing message to which a response is expected. Such a process transition runs to completion (as always) and the process goes into "Inactive" mode. When the expected reply message is received, it goes into the input buffer ring and is handled by activating the original process in the usual manner.

Since it has been assumed messages may possibly be lost, a facility for timers to detect such lost messages is needed. One possible general rule is that a process which is waiting for a reply to a message should use a timer to detect a lost message. Such a process is said to be in a "transient state" , and if either the original message or the response has been lost the process could otherwise remain in such a state indefinitely. The expiration of the timer sends a timeout message to trigger some sort of recovery action by the process. The sort of recovery action which may be employed depends on the sort of processes that are involved and the nature of the failure. For example, in an exchange between "persistent" processes (those with backups or which are automatically recreated), the action may be to check the status of the partner (by sending a message to it), or to rebind its name, and continue until successful. For ephemeral processes (such as those that manage call setups or transactions) the action may be simply to abort, with any retry generated by external (to those processes) means.

In such a system, in which many processes go into transient states and almost always leave them very soon, it is important that the costs of both setting and cancelling timers is kept very low. We are assuming here that the timeout values are somewhat arbitrary; provided that they are much greater than the normal response time, the actual value of the timeout is not especially important, and there is usually no requirement that timeout delays are honoured precisely.

In these circumstances, a very low cost implementation might use a data structure of the form:

```
timer := record
              link(s):  ^timer;
              owner:    process_id
              value:    integer;
         end;
```

Timers can be created and destroyed by a process using the following two procedures; the timer pointer(s) are kept in the process state.

```
Create_timer () → ^timer
```

This process creates a new timer record and returns its pointer; the owner field gets the id of the currently running process and the timer record is appended to a global timer chain. (If different clock resolutions are needed, then the resolution becomes an input parameter). These timer chains are global variables managed by the kernel.

```
Destroy_timer( t:^timer)
```

This deletes a timer from the global timer chain, (This needs a double-linked chain for efficiency), and then deallocates it.

Timer setting and cancelling may then be done very cheaply by the following two macros:

```
set_timer( t:^timer, v;integer) {= t^.value ← v}
cancel_timer(t:^timer) {t^.value ← -1}
```

The real time clock interrupt periodically arranges to run a timer scan (between application process executions) that runs through all the chained timers, decrementing any non-negative value fields. Any value field which becomes 0 results in a message being sent to the owner process.

The timeout message here is assumed to have the command "timeout" so the resulting action is purely state driven. Alternatively, the set_timer can include a message code for use on the timeout message.

Numerous modifications and variations of the present

invention are possible in light of the above teachings. It is therefore to be understood that within the scope of the appended claims, the invention may be practised otherwise than as specifically described herein.

While a specific mechanism for achieving atomicity of state transitions has been described, it is to be understood that other mechanisms may be used.

Process IDS (pid) may be used to represent capabilities. Having a pid entitles an entity to send whatever messages the associated process will accept and hence represents a capability to do certain things. In this described embodiment pids are just binary values and can be stored in any convenient data structure and passed freely in messages, etc., and there are no controls or restrictions placed on such capability passing. In such a system, revoking a capability previously granted can only be done by destroying the process and then creating a new one with a different pid to carry on its function if necessary; node reinitialization being an extreme case of this. The working assumption is that this is a benign cooperative environment, with all nodes "trusted" to be non-malicious, but possibly misbehaving under failure conditions.

Incarnation numbers, which fill the otherwise unused bits of the pid, provide that the same process id is not reused within some time interval, sufficient that old memorized pids will likely have been purged. A cheap way to revoke outstanding capabilities is then to increment the incarnation number of a process while maintaining its state.

Of course, implementations without incarnation numbers are possible.

WE CLAIM:

1. In a network node having a processor, and a memory, an asynchronous message processing method comprising the steps of:

maintaining a plurality of processes in the memory, each process consisting of a process state and associated application code, each process being identifiable by a pid (process identifier);

the node receiving incoming messages;

for each incoming message received, the processor determining from a destination pid forming part of the message to which of said plurality of processes the message is addressed, and running the application code associated with the destination pid to handle the message as a function of the destination process's process state and making changes to the destination process's process state if necessary;

the application task, while running, generating outgoing messages if necessary;

the node transmitting any outgoing messages.

2. A method according to claim 1 further comprising the steps of:

an interface forming part of the node receiving the incoming messages and storing them in an input message queue in the order they are received;

wherein the processor reads messages stored in the input message queue and processes them one at a time.

3. A method according to claim 2 further

comprising the steps of:

the application code storing the outgoing messages in an output message queue in the order they are generated; and

the interface reading messages from the output message queue and transmitting them one at a time.

4. A method according to claim 1 wherein each process has a process type definition stored in memory which identifies the associated application code.

5. A method according to claim 4 wherein to create an instance of a new process the processor allocates in the memory a new respective process state.

6. A method according to claim 1, wherein the process state comprises a plurality of state variables having previous values which existed prior to the start of the application code, the method further comprising the steps of:

the processor logging the previous value of any state variables modified by the application code during the transistor;

the application code finishing in either a commit, an abort or an exception; and

the processor, in the case of the application code finishing with either an abort or exception, restoring the state variables modified to their previous values.

7. A method according to claim 6 wherein the step of logging the previous value of any state variables

modified by the application code is achieved by the processor storing each previous value and its address in memory to a location in a trail stack indicated by a trail stack pointer and then incrementing the trail stack pointer; wherein

in the case of the application code finishing in an commit, the processor resets the trail stack pointer; and

in the case of the application code finishing in an abort or exception, the processor restoring each previous value in the trail stack to its original address in memory.

8. A method according to claim 6 wherein the messages stored in the output queue by the application code are only transmitted in the case that the application code finishes in a commit.

9. A method according to claim 1 wherein each incoming message contains a source pid which identifies a process which sent the message.

10. A method according to claim 3 wherein the input message queue and the output message queue are stored in the memory and are accessible by the processor, and accessible by direct memory access by the interface and all necessary normal communication and control between processes and the interface is solely through these queues.

11. A method according to claim 1 wherein a process

instance table is used to map each pid with a location in memory of the corresponding process state and with a location in memory of the application code.

12. A method according to claim 1 further comprising the steps of:

when a message is generated by a particular process and subsequently transmitted by the node, and the particular process expects a response incoming message in return, the process starting a timer, and the kernel decrementing it periodically; and

when the timer expires, the processor sending a timeout message to the particular process.

13. A network node for connection to a digital network comprising:

an interface to the digital network for receiving incoming messages and storing them in an input message queue, and for reading outgoing messages from an output message queue and transmitting them;

a processor for maintaining a plurality of processes in a memory, each process consisting of a process state and associated application code, each process being identifiable by a pid (process identifier), and for reading messages one at a time from the input message queue in the order they were received, determining from a destination pid forming part of the message to which of said plurality of processes the message is addressed, and running the application code associated with the destination pid to handle the message as a function of the destination process's process state and making changes to the destination process's process state if necessary and writing an outgoing message to a next available buffer location in the output message queue if necessary.

14. A digital network comprising a plurality of network nodes, each network node comprising:

an interface to the digital network for receiving incoming messages and storing them in an input message queue, and for reading outgoing messages from an output message queue and transmitting them;

a processor for maintaining a plurality of processes in a memory, each process consisting of a process state and associated application code, each process being identifiable by a pid (process identifier), and for reading messages one at a time from the input message queue in the order they were received, determining from a destination pid forming part of the message to which of said plurality of processes the message is addressed, and running the application code associated with the destination pid to handle the message as a function of the destination process's process state and making changes to the destination process's process state if necessary and writing an outgoing message to a next available buffer location in the output message queue if necessary.

15. A digital network according to claim 14 further comprising a process name server which knows the pids of some processes running on each node, wherein for a first of said plurality of network nodes to identify a process

on a second of said plurality of network nodes, the first node sends a query to the process name server which responds with the process identifier of the process on the second node.

FIG. 1
NETWORK CONTEXT

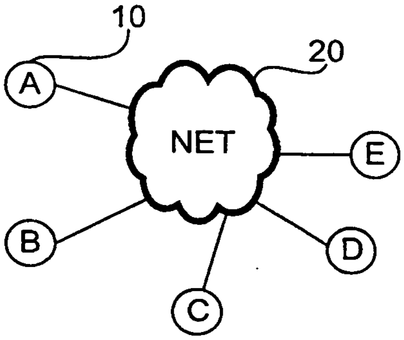


FIG. 2
NODE HARDWARE

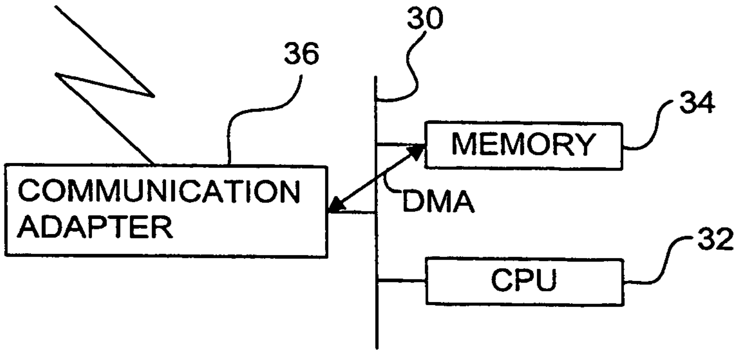


FIG. 3
OVERALL MESSAGE FLOW

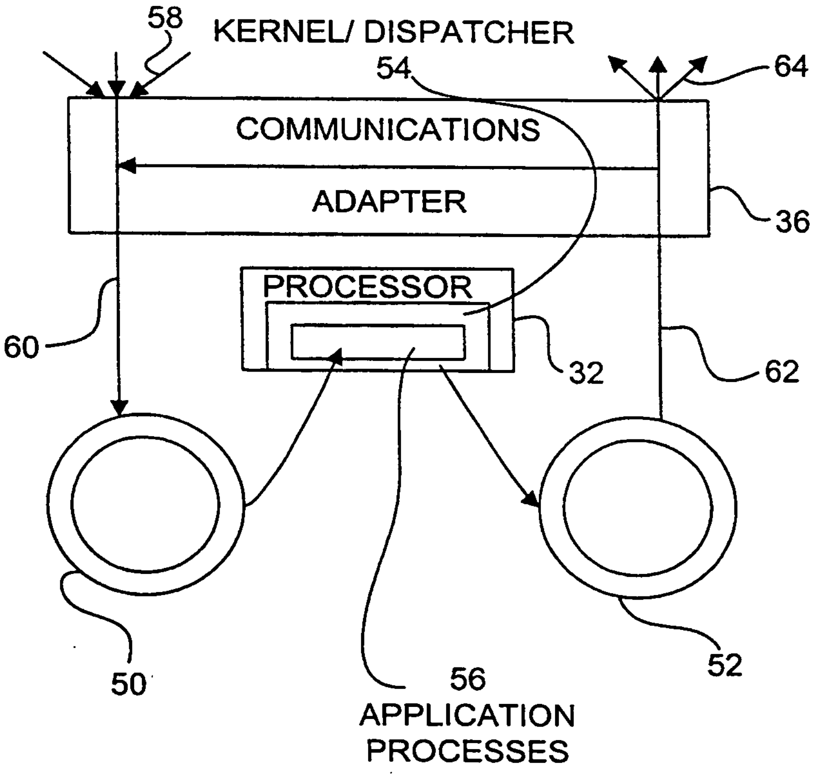
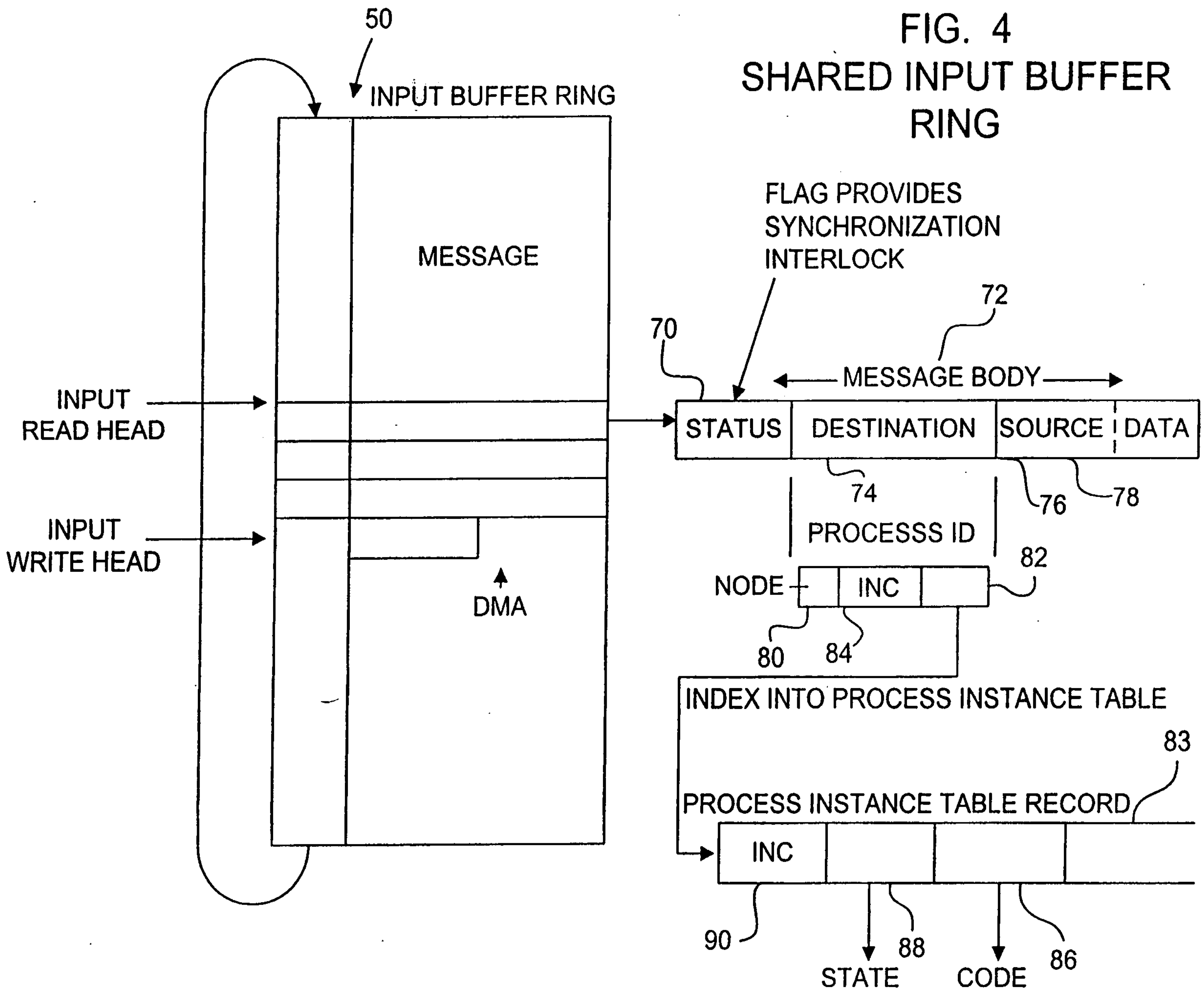


FIG. 4
SHARED INPUT BUFFER RING



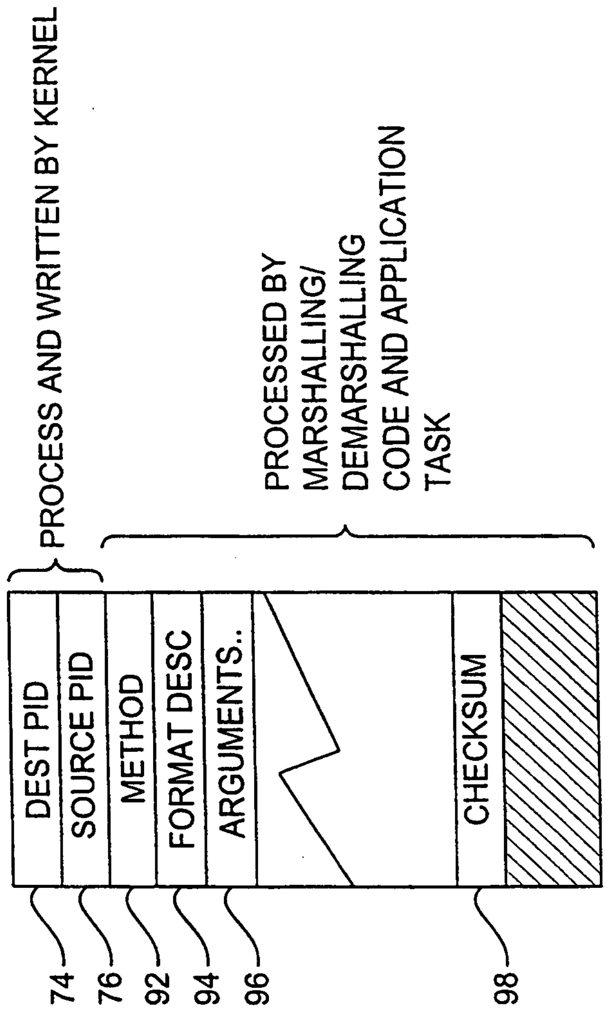
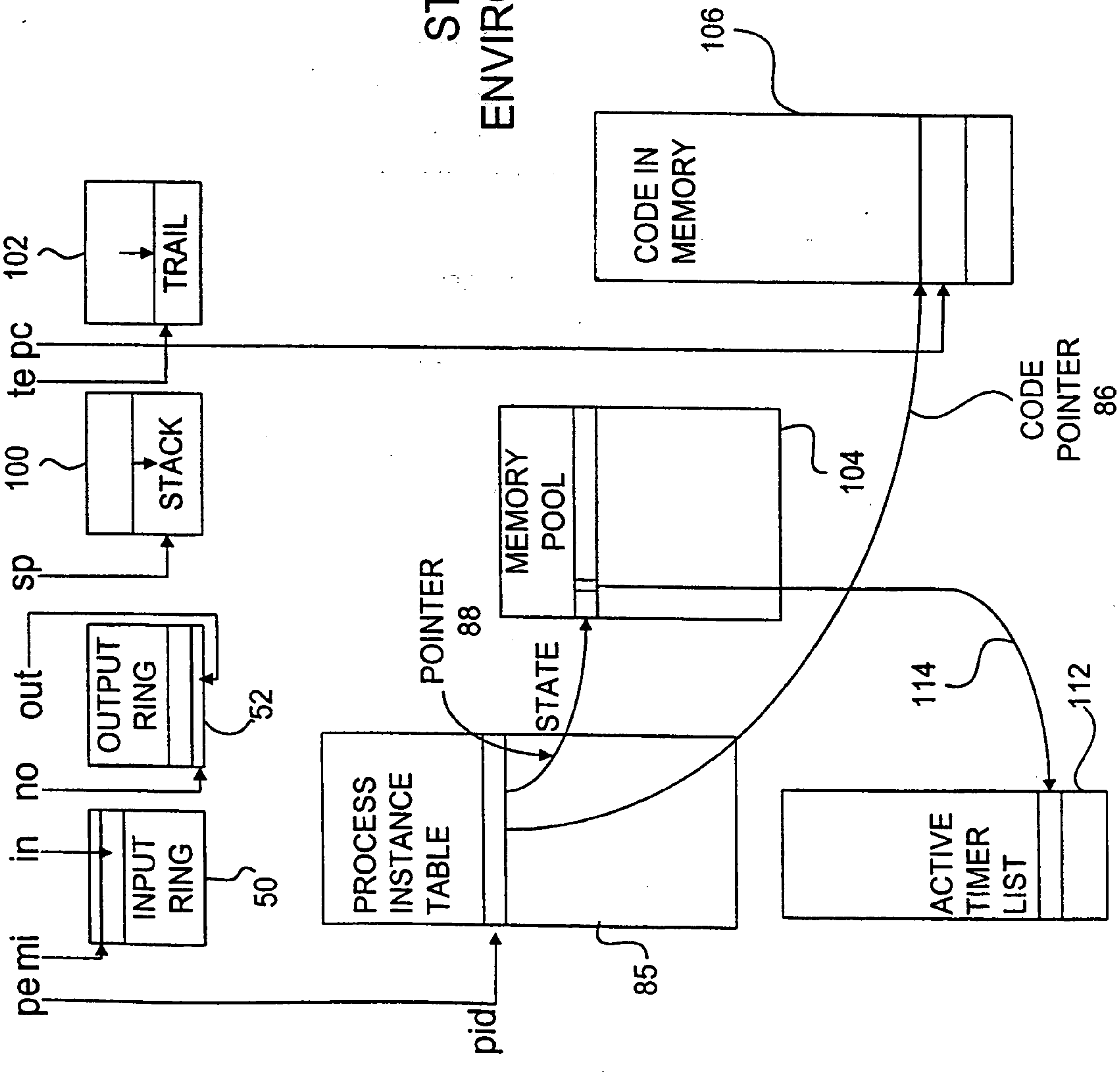

PROCESS AND WRITTEN BY KERNEL
PROCESSED BY MARSHALLING/ DEMARSHALLING CODE AND APPLICATION TASK
DEST PID
SOURCE PID
METHOD
FORMAT DESC
ARGUMENTS..
CHECKSUM
74
76
92
94
96
98

FIG. 5
POSSIBLE MESSAGE STRUCTURE

FIG. 6
KERNEL DATA
STRUCTURES AND
ENVIRONMENT REGISTERS



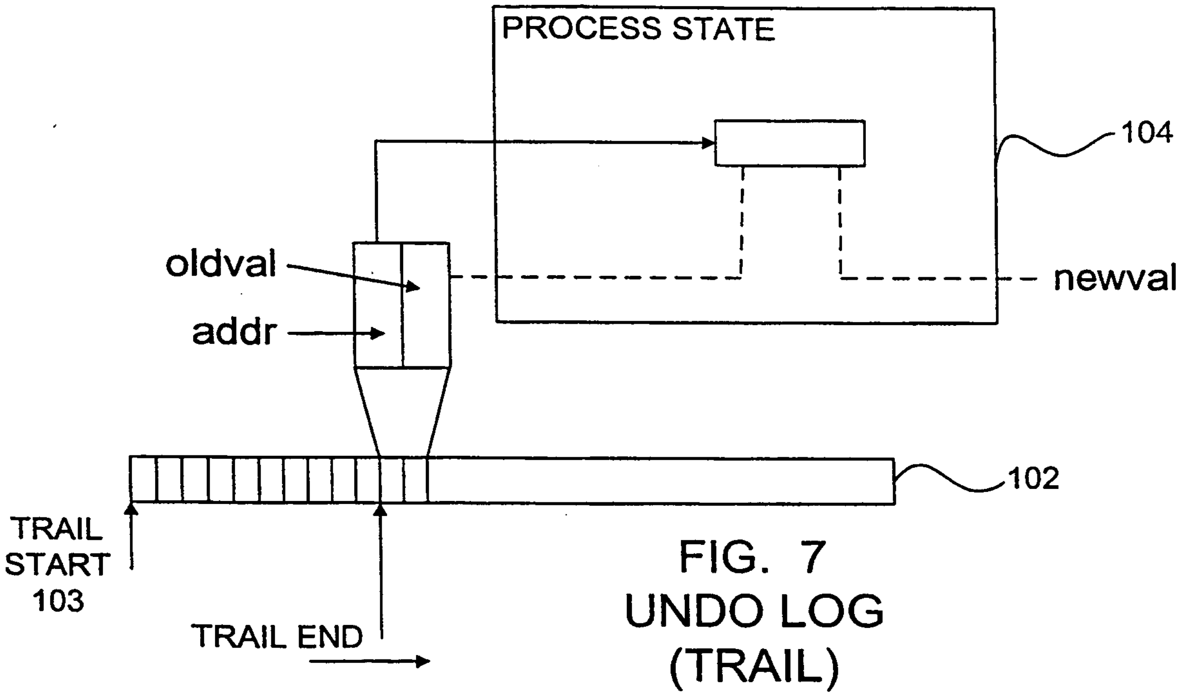
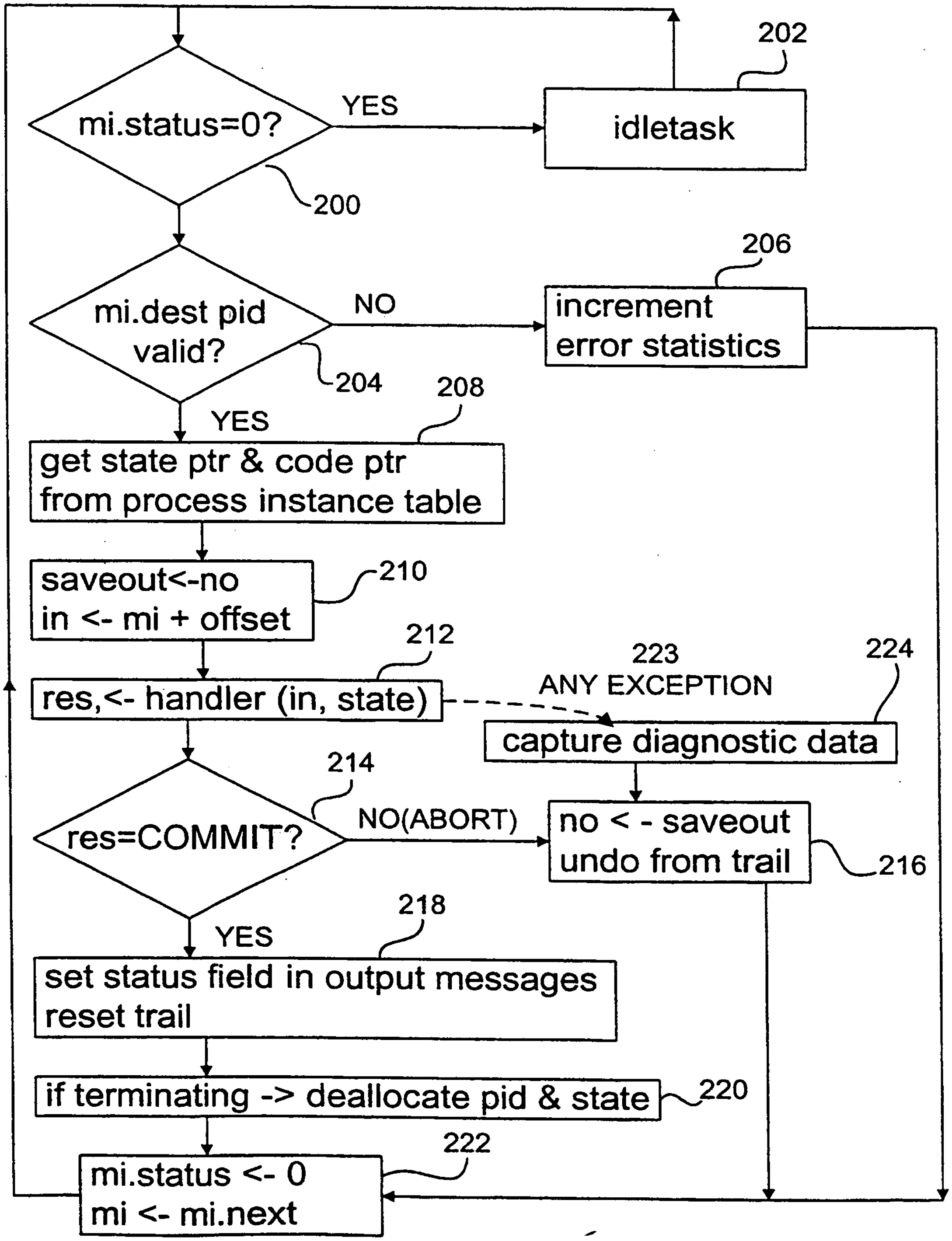


FIG. 7
UNDO LOG
(TRAIL)
MECHANISM

FIG. 8
KERNAL DISPATCHER
LOOP



PCT

WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 6 :
H04L 29/06

A3

(11) International Publication Number: WO 98/59519

(43) International Publication Date: 30 December 1998 (30.12.98)

(21) International Application Number: PCT/CA98/00537

(22) International Filing Date: 2 June 1998 (02.06.98)

(30) Priority Data:
08/880,459 24 June 1997 (24.06.97) US

(71) Applicant: NORTHERN TELECOM LIMITED [CA/CA]; P.O. Box 6123, Station A, Montreal, Quebec H3C 3J5 (CA).

(72) Inventors: OLDER, William, Joseph; R.R. # 3, Merrickville, Ontario K0G 1N0 (CA). BIERMAN, Eric; 1701-71 Somerset Street W., Ottawa, Ontario K2P 2G2 (CA).

(74) Agents: BRETT, R., Allan et al.; Smart & Biggar, 900-55 Metcalfe Street, P.O. Box 2999, Station D, Ottawa, Ontario K1P 5Y6 (CA).

(81) Designated States: CA, JP, European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE).

Published
With international search report.
Before the expiration of the time limit for amending the claims and to be republished in the event of the receipt of amendments.

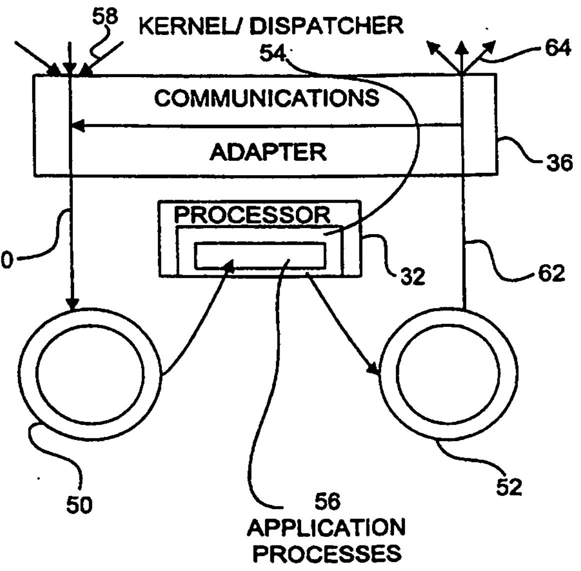
(88) Date of publication of the international search report:
25 March 1999 (25.03.99)

(54) Title: ASYNCHRONOUS MESSAGE PROCESSING SYSTEM AND METHOD

(57) Abstract

A new asynchronous message processing system and method are provided in which a process is completely defined by a state and a type. Incoming messages received by a network node are stored in an input message queue and are handled in sequence in the order they arrive, and outgoing messages are stored in an output message queue and are transmitted in sequence in the order they are generated to a destination network node. Incoming messages addressed to a process are handled by invoking a particular message handling application code which runs to completion, and which may modify the process state. Process state modifications are atomic in the sense that they either completely happen or do not happen at all; partially implemented process state modifications are undone.

OVERALL MESSAGE FLOW



FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AL	Albania
AM	Armenia
AT	Austria
AU	Australia
AZ	Azerbaijan
BA	Bosnia and Herzegovina
BB	Barbados
BE	Belgium
BF	Burkina Faso
BG	Bulgaria
BJ	Benin
BR	Brazil
BY	Belarus
CA	Canada
CF	Central African Republic
CG	Congo
CH	Switzerland
CI	Côte d'Ivoire
CM	Cameroon
CN	China
CU	Cuba
CZ	Czech Republic
DE	Germany
DK	Denmark
EE	Estonia
ES	Spain
FI	Finland
FR	France
GA	Gabon
GB	United Kingdom
GE	Georgia
GH	Ghana
GN	Guinea
GR	Greece
HU	Hungary
IE	Ireland
IL	Israel
IS	Iceland
IT	Italy
JP	Japan
KE	Kenya
KG	Kyrgyzstan
KP	Democratic People's Republic of Korea
KR	Republic of Korea
KZ	Kazakstan
LC	Saint Lucia
LI	Liechtenstein
LK	Sri Lanka
LR	Liberia
LS	Lesotho
LT	Lithuania
LU	Luxembourg
LV	Latvia
MC	Monaco
MD	Republic of Moldova
MG	Madagascar
MK	The former Yugoslav Republic of Macedonia
ML	Mali
MN	Mongolia
MR	Mauritania
MW	Malawi
MX	Mexico
NE	Niger
NL	Netherlands
NO	Norway
NZ	New Zealand
PL	Poland
PT	Portugal
RO	Romania
RU	Russian Federation
SD	Sudan
SE	Sweden
SG	Singapore
SI	Slovenia
SK	Slovakia
SN	Senegal
SZ	Swaziland
TD	Chad
TG	Togo
TJ	Tajikistan
TM	Turkmenistan
TR	Turkey
TT	Trinidad and Tobago
UA	Ukraine
UG	Uganda
US	United States of America
UZ	Uzbekistan
VN	Viet Nam
YU	Yugoslavia
ZW	Zimbabwe

INTERNATIONAL SEARCH REPORT

Inte ional Application No
PCT/CA 98/00537

A. CLASSIFICATION OF SUBJECT MATTER
IPC 6 H04L29/06

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
IPC 6 H04L

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practical, search terms used)

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category °	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X	US 4 736 364 A (BASSO RICHARD J ET AL) 5 April 1988 see abstract see column 6, line 39 - column 8, line 5 see figure 1 ---	1-4,10, 11,13,14
A	N THEURETZBACHER: "ENHANCED CHILL TASKING CONCEPT AND LANGUAGE FOR A BUSINESS COMMUNICATION SYSTEM" ELECTRICAL COMMUNICATION, vol. 58, no. 2, 1983, pages 213-217, XP002082834 -----	1-15

□ Further documents are listed in the continuation of box C.

☒ Patent family members are listed in annex.

° Special categories of cited documents :

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier document but published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art.

"&" document member of the same patent family

Date of the actual completion of the international search 27 January 1999	Date of mailing of the international search report 05/02/1999
Name and mailing address of the ISA European Patent Office, P.B. 5818 Patentlaan 2 NL - 2280 HV Rijswijk Tel. (+31-70) 340-2040, Tx. 31 651 epo nl, Fax: (+31-70) 340-3016	Authorized officer Canosa Areste, C

INTERNATIONAL SEARCH REPORT

Information on patent family members

Inte onal Application No
PCT/CA 98/00537

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
US 4736364 A	05-04-1988	CA 1277041 A	27-11-1990
		DE 3786184 A	22-07-1993
		DE 3786184 T	05-01-1994
		EP 0237247 A	16-09-1987
		JP 1929371 C	12-05-1995
		JP 6057076 B	27-07-1994
		JP 62230296 A	08-10-1987
		KR 9513171 B	25-10-1995